Attempts to
Solving Traveling Salesman Problem
using Hopfield neural network

Alireza Mahmoudian[1]

[1]SID: 95112095

Chapter 1

Introduction

1.1 Traveling Salesman Problem

TSP is a famous NP-complete problem which has both theoretical and practical importance. Given a directed weighted graph, the problem is to find a sequence of nodes that form a Hamiltonian path of the minimum cost. From theoretical point of view it's importance is because of its NP-completeness, which means solving of many other NP problems in tractable time depends on it's solvability. It's practical importance arises from it's application, from logistics and transportation (Of course, concerning more than just a traveling salesman[1]) to designing of PCBs[2].

What makes TSP a special case is that, unlike many other NP-complete optimization problems, it has no approximation algorithm theoretically. An approximation algorithm for an optimization problem, is an algorithm that guaranties to obtain a proportion α of the optimization result.[3] It can be proved that not only TSP itself is NP-complete, but also any α-approximator of it is also NP-complete. So it would be interesting to talk about approximating such problem.[4]

1.2 Hopfield Network

Hopfield neural network is a fully connected recurrent network that can act as associative memory. We see patterns that are taught to the Hopfield network as particles in a dynamical system. So as system reaches it's stable state, particles reach their minimum energy level. That's how an associative memory works; If we feed a new particle to the system, it would find the nearest state that makes it's energy level minimum, which is the state that was taught to the system before.

But here, we don't need associative memory, we rather want to employ another interesting trait of the Hopfield network; Its ability of optimization. So all we have to is to define this error surface as the objective function that we want to optimize. Then gradient descent approach to adjust network weight so that it converges to a stable state.

[1] A new promising method is proposed here: https://xkcd.com/399/

[2] Printed Circuit Board

[3] $\alpha > 1$ for minimization and $0 < \alpha < 1$ for a maximization problem. So an approximation algorithm can tell us how far we are from the global optimum.

[4] Of course, a neural network is not an approximation algorithm; In the first place, it does not have the "guaranties" constraint mentioned in the definition. It also won't tell us anything about how close we are from the global optimum.

1.3 Energy Function

For TSP, want we want to optimize is the cost of a spanning rout. So we have to define Hopfield's energy function based on the sum of edge's weights that are in a rout. In order to show a state of a particle as a rout in TSP, we define the network as a matrix of fully connected Hopfield neurons $N_{n\times n}$, where n is the size of the TSP graph, such that in a state k of the network, $N_{xi}^k = 1$ would indicate that the node x of the graph is met at the step i of the tour. With this scheme, definition of TSP can be adapted to our notion of Hopfield network. We can then define energy function such that it represents the TSP's objective function. Assume that the edge weights are stored in an adjacency matrix Δ, such that Δ_{xy} be the weight of the edge between node x and y. Then we want to find an optimal N where the following term is minimum:

$$\sum_{x=1}^{n}\sum_{y=1}^{n}\sum_{i=1}^{n}\Delta_{x,y}N_{x,i}(N_{y,\ i\ +_n 1} + N_{y,\ i\ -_n 1})$$

But it can be easily seen not all minimum states of N are feasible TSP solutions. In fact it also takes some state which there are duplicate nodes, or steps with more than one node met. So we have to set constraints to prevent such states.

The problem here is that optimization constraints are not something that Hopfield networks are meant to handle. All we have is the energy function. So we have to redefine the energy function, forcing the network itself to avoid the invalid states.

This can be done by relaxation of constraints. We add more terms to the energy function which would increase its value if some invalid state is taken, thus penalizing the network. So our energy function will be like this:

$$\begin{aligned}\mathcal{E} = {} & D\sum_{x=1}^{n}\sum_{y=1}^{n}\sum_{i=1}^{n}\Delta_{x,y}N_{x,i}(N_{y,\ i\ +_n 1} + N_{y,\ i\ -_n 1}) \\ & +\ A\sum_{x=1}^{n}\sum_{i=1}^{n}\sum_{j=1,j\neq i}N_{x,i}N_{x,j} \\ & +\ B\sum_{i=1}^{n}\sum_{x=1}^{n}\sum_{y=1,y\neq x}N_{x,i}N_{y,i} \\ & +\ C(\sum_{x=1}^{n}\sum_{i=1}^{n}N_{x,i} - n)^2\end{aligned} \tag{1.1}$$

The second term would have a value bigger than zero, only if there are rows with more than one 1s, which means that the node sequence has duplicates. So as it's increment is penalized, this error would be avoided. Similarly, the third term controls if there are more than one 1s in each column. And the fourth term checks if the number of all neurons in state 1 is exactly n, to avoid states with no 1 in some rows or columns.

Notice that A, B, C and D are constant factors to control the effect of each constraint on the total energy. They are meta-parameters and should be estimated by user for different problem instances.

1.4 Synaptic weights adjustment

We want energy function to decrease over time, so let $\frac{d\mathcal{E}}{dt} < 0$. But we can't calculate weights directly from $\mathcal{E}$. So we use chain rule so that $\frac{d\mathcal{E}}{dt} = \frac{\partial\mathcal{E}}{\partial N_{x,i}} \times \frac{dN_{x,i}}{dnet_{x,i}} \times \frac{dnet_{x,i}}{dt}$. We know

that $\frac{dN_{x,i}}{dnet_{x,i}} > 0$, since it is derivative of the activation function and it is (sigmoid or tanh) strictly increasing. But don't have control over $\frac{\partial \mathcal{E}}{\partial N_{x,i}}$, and it could be positive or negative. But we can set $\frac{dnet_{x,i}}{dt} = -\frac{\partial \mathcal{E}}{\partial N_{x,i}}$, so that we can be sure that $\frac{d\mathcal{E}}{dt} = -\frac{\partial \mathcal{E}}{\partial N_{x,i}}^2 \times \frac{dN_{x,i}}{dnet_{x,i}}$ is negative. So we'll have:

$$\begin{aligned}
\frac{dnet_{x,i}}{dt} &= -\frac{\partial \mathcal{E}}{\partial N_{x,i}} \\
&= -2A \sum_{j \neq i} N_{x,j} \\
&\quad - 2B \sum_{y \neq x} N_{y,i} \\
&\quad - 2C(\sum_{y} \sum_{j} N_{y,j} - n) \\
&\quad - 2D \sum_{x \neq y} \Delta_{x,y}(N_{y,\ i\ +_n 1} + N_{y,\ i\ -_n 1})
\end{aligned} \tag{1.2}$$

That is how neurons' input should change over time, and since it is a function of synaptic weights. Notice that A, B, C and D factors are multiplied by 2 after derivation, and since they are just constants, we can give them new names and omit 2 factors.

Chapter 2

Experiments

2.1 The Problem and Goals of Experiment

In these experiments, the methods that was described in the previous chapter were employed to reach satisfying results for TSP. This was done by searching for meta-parameters A, B, C and D for different benchmarks. As it was discussed in previous reports, this process is a [blindly] search for values for these meta-parameters with which the network gives the best results.

For discriminating these "best" results, they were compared with some benchmark results, but before that, we should check for feasibility of each result, since it's very much probable that the relaxed constraint of the energy function were defied. So each run of the algorithm was followed by a Hamiltonian cycle check.

We evaluated results on three different benchmarks whose optimal TSP distance was calculated before by other means. So we introduced α, a proportion of our network's result and the benchmark result at hand. Obviously the closer it is to 1, the better the result is.

2.2 Benchmarks

Benchmark results that are used here, were provided via TSPLIB[1], "a library of sample instances for the TSP (and related problems) from various sources and of various types". Three datasets that were used, consist of real geographical coordinations[2]. of cities in various countries. Thus they were first converted to adjacency matrix. The datasets we used can be found with their respective solutions in the mentioned library by names `ulysses16`, `ulysses22` and `gr24`. As their names suggest, they are coordinates of 16, 22 and 24 nodes, respectively. Euclidean distance was used to extract adjacency matrix for them.

2.3 Methodology

This is a briefing of what our program does.

Function f takes A, C, D and adjacency matrix Δ as input parameters and perform optimization for TSP over Δ, and repeats it 10 times exactly, via the method that was

[1] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

[2] I just want to mention in fact there IS approximation algorithm for metric TSPs, like these ones.

described in the previous chapter. It check each result for feasibility (i.e. being a Hamiltonian path) and calculates average of the cost of feasible results. It then prints its output in a file. It assumes A=B, as it was recommended by literature, experience and intuition.

In the main program, f is called inside three loops for A, C, and D. These loops traverse a portion of infinitely huge space for these parameters, of course, by user interaction when ever it is needed.

2.4 Remarks

- A few tries assured me that some little difference between A and B values could lead to the bigger parameter outdoing the lesser one and at least one of their respective constraints were defied. So as it was recommended by some previous experiments, as well as Hopfield's, I assumed $A = B$ afterwards. It is also confirms out intuition, since their constraints are alike in many aspects.

- The meta-parameters, A, B, C, and D are just defining a measure for four terms in the energy function. We know that without any change in the optimal result, we can multiply any other value in the energy function. So we can even eliminate more of these meta-parameters, via dividing the whole function by one of them.

- There were some recommendations about normalizing the dataset, which I had guessed, and saw in the experiments, that was not helpful. In fact, as mentioned, these parameters in fact are doing a normalization.

- There was an error in the definition of the energy function that was presented in the class, which ~~surprisingly~~[3] no body seemed to have any problem with that in the implementation. The slide says that the negation of parameter D, multiplied by $\Delta_{x,y}$[4] is added to the weight "between nodes in different row but adjacent column", which are demonstrated by column indices i+1 and i-1, so it doesn't take into account that, the distance between last node and the first node is also involved in calculating the tour's cost, because it's gotta be a Hamiltonian tour, not a path. I used modular addition and subtraction in my implementation. So in this notion, the last column of neurons would be adjacent to the last column.

- At first I used Neuroph, the same library I had used for MLP and RBF. But soon I found that it's too much for this purpose. The overhead of using the library, like adapting the matrix schema of the network to the library's layered architecture made me leave it after a while. It was just matrix calculations and I could see that even completely out of the neural networks perspective. So I did it all from the scratch. In the new code there were no sign of neural nets, Just loops over matrices.

- Similar to previous reports about tuning up MLP and RBF, Here we made an exhaustive search, starting from recommended settings (By some researchers, as well as Hopfield), search in more details in interesting spots.

- Each Hopfield network recurrence was made asynchronously, each time with a random order.

[3] Well, who cares, Matlab can handle every thing automagically for you anyways.
[4] Called "d_{xy}" at the context.

Table 2.1: Benchmark 1: `ulysses16`

A,B	C	D	α
126000	2352	8	0.8520489646
78000	1556	5	0.8293456247
130000	2460	8	0.8254031057
110000	2220	7	0.8225382082
85000	1870	5	0.8182447966
63000	1426	4	0.8116441869
103000	2606	7	0.8043081997
62000	1724	4	0.7992448128
115000	2330	8	0.7977358863
125000	2350	8	0.7936070439
131000	3062	9	0.7914454554
126000	2852	9	0.7845816454
127000	2854	8	0.7844995541
123000	2446	8	0.7817233318
130000	2860	9	0.7816969756
105000	2110	7	0.780958781
104000	2708	7	0.7773008079
106000	3012	8	0.7772289508
120000	2840	8	0.7769812713
116000	2632	9	0.7752910498

- An evaluation parameter α was introduced as the ratio of the optimal benchmark result to our result from the network. It shows how much the network could get near the real solution.
- The source code for this project, including this report, can be found at `https://github.com/BelegCuthalion/hopfield-tsp`

2.5 Results

Tables 1, 2 and 3 are showing the best (feasible) results, by means of it's proximity to the real optimal value, for different benchmarks.

It can be seen that an increase in the problem size had so much damage to the functionality of the network. With the first benchmark that has 16 nodes, it reached as good as 85% of the optimal value. But it dropped below 50% when the problem size increased to 22. Larger graphs were also used in experiments, but the results was disappointing enough to convince us not to continue. So the TSP size must have a great effect on the functionality of the method.

Also notice the proportion of the parameters A (=B), C and D. They are in sort of a fixed ratio (at least logarithmically). And even when the distances were normalized, it was enough to change these parameters at the same rate to obtain similar results.

It worths to mention that these results are a portion of nearly a hundred thousand different settings (also counting the infeasible results) for meta-parameters.

Table 2.2: Benchmark 2: `ulysses22`

A,B	C	D	α
110500	3521	7	0.4924114252
30500	761	2	0.4910209576
500000	7700	26	0.4908883267
500000	5000	29	0.4846873656
70500	2841	1	0.4792191663
1000000	10400	59	0.478330383
120500	2941	9	0.477677088
100500	3201	5	0.4774347773
1000000	10300	55	0.4774216009
1000000	10300	57	0.4766860545
130500	2561	9	0.4756850182
70500	2141	4	0.4755165606
120500	3041	8	0.474486513
1000000	10100	48	0.4736397011
130500	3461	7	0.4732877725
120500	2941	7	0.472947914
1000000	10000	62	0.4727467808
120500	3741	5	0.4726533738
130500	4261	5	0.4725186626
110500	2521	7	0.4720275141

Table 2.3: Benchmark 3: `gr24`

A,B	C	D	α
10297	502	4	0.4256016864
200000	2300	83	0.4184836933
400000	4600	162	0.4141464028
500000	4900	187	0.4123220523
12970	629	7	0.4122217341
200000	1700	64	0.4108604578
12178	621	5	0.4107365999
200000	2200	78	0.4096241183
12376	623	6	0.4086401408
200000	1700	50	0.4064823364
11089	510	3	0.4062184701
100000	1300	40	0.4057309972
300000	2200	11	0.4043110769
400000	3800	132	0.4033860459
200000	2200	77	0.4029960906
300000	3600	119	0.4025931543
100000	1700	41	0.4021731867
400000	4100	140	0.4018252478
12574	225	5	0.40109603
500000	2300	38	0.4010588445
10198	201	2	0.4005134801